



may be defined as being derived from another class that is referred to as its "base class." The derived class inherits from the base class in that an object of the derived class automatically gets the data and method members of the base class. The derived class may refine the concept represented by the base class by defining additional data or method members or redefining method members to override those of the base class. Class inheritance provides the powerful intellectual tool of hierarchical ordering for managing the complexity of a program. In this regard, a program can often be organized as a set of trees or directed acyclic graphs of classes. Each node of the tree may be a class derived from another class and may itself be the base class for another derived class.

In many software development projects, a software application is constantly being refined over the life span of the project. Through the development process, concepts concerning various problems and solutions are often revised many times, and the functions and features of the final software application are often quite different from those defined at the beginning of the project. When an object-oriented programming language, such as the C++ language, is used to develop the software, the revisions of concepts would result in the introduction of new classes. In this regard, it is common for program developers to start with a set of broadly defined base classes, and incrementally refine the concepts by designing new classes derived from the existing

5

10

Thus, the introduction of a new derived class may require extensive editing and recompilation of existing source listing

originally programmed to create objects of the old base classes. This process can be very time consuming and prone to errors, especially when the software project is large and the class hierarchy is complex. As a result, existing object-oriented programming languages do not facilitate incremental development of a software project by gradually replacing base classes with new derived classes.

#### SUMMARY OF THE INVENTION

10 In view of the foregoing, the present invention provides a new approach and language extension to object-oriented programming that uses replaceable classes and virtual constructors to enable the reuse of existing code written for creating objects of a base class to dynamically create objects  
15 of a new class inheriting from the base class. The class replacement can be implemented in both compiled and interpreted languages, but its advantages are especially significant when the source code containing the definitions of the replaceable classes is compiled to form a reusable module.

20 In accordance with the invention, a class that is intended to be replaceable later is declared in the source code as being a replaceable class. In subsequent software development, a replacement class derived from the replaceable class may be defined in new source code, and an instruction to  
25 replace the base class with the new derived class during program execution is included in the new source code. In the case where the programming language is a compiled language,

0053990-033000

5

10

15

20

BRIEF DESCRIPTION OF THE DRAWINGS

While the appended claims set forth the features of the present invention with particularity, the invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

FIG. 1 is a block diagram generally illustrating an exemplary computer system on which the software development utilizing replaceable classes in accordance with the invention may be performed;

FIG. 2 is a schematic diagram showing a computer program with exiting modules that generate a memory structure of objects of given classes and a modified memory structure that is intended to be generated to contain objects of later developed new classes;

FIG. 3 is a schematic diagram showing a computer program that uses a reusable module with old classes to generate objects of new classes in accordance with the invention;

FIG. 4 is a schematic diagram showing the compilation of exemplary source listings into respective object (.obj) files with data emitted therein for supporting class replacement in accordance with an embodiment of the invention;

FIG. 5 is a schematic diagram showing memory images of objects of two base classes and a replacement class that inherits from the two base classes;

FIG. 7 is a schematic diagram showing the data emitted into the initialized data segment of the .obj file for the replacement class from FIG.4 for supporting class replacement; and

## DETAILED DESCRIPTION OF THE INVENTION

The invention may also be practiced in distributed computing

5 With reference to Fig. 1, an exemplary system for  
implementing the invention includes a general purpose  
computing device in the form of a conventional personal  
computer 20, including a processing unit 21, a system memory  
22, and a system bus 23 that couples various system components  
10 including the system memory to the processing unit 21. The  
system bus 23 may be any of several types of bus structures  
including a memory bus or memory controller, a peripheral bus,  
and a local bus using any of a variety of bus architectures.  
The system memory includes read only memory (ROM) 24 and  
15 random access memory (RAM) 25. A basic input/output system  
(BIOS) 26, containing the basic routines that help to transfer  
information between elements within the personal computer 20,  
such as during start-up, is stored in ROM 24. The personal  
computer 20 further includes a hard disk drive 27 for reading  
20 from and writing to a hard disk 60, a magnetic disk drive 28  
for reading from or writing to a removable magnetic disk 29,  
and an optical disk drive 30 for reading from or writing to a  
removable optical disk 31 such as a CD ROM or other optical  
media.

25       The hard disk drive 27, magnetic disk drive 28, and  
optical disk drive 30 are connected to the system bus 23 by a  
hard disk drive interface 32, a magnetic disk drive interface



33, and an optical disk drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for the personal computer 20. Although the exemplary environment described herein employs a hard disk 60, a removable magnetic disk 29, and a removable optical disk 31, it will be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories, read only memories, and the like may also be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk 60, magnetic disk 29, optical disk 31, ROM 24 or RAM 25, including an operating system 35, one or more applications programs 36, other program modules 37, and program data 38. A user may enter commands and information into the personal computer 20 through input devices such as a keyboard 40 and a pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23

5       The personal computer 20 may operate in a networked  
environment using logical connections to one or more remote  
computers, such as a remote computer 49. The remote computer  
49 may be another personal computer, a server, a router, a  
network PC, a peer device or other common network node, and  
10 typically includes many or all of the elements described above  
relative to the personal computer 20, although only a memory  
storage device 50 has been illustrated in Fig. 1. The logical  
connections depicted in Fig. 1 include a local area network  
(LAN) 51 and a wide area network (WAN) 52. Such networking  
15 environments are commonplace in offices, enterprise-wide  
computer networks, intranets and the Internet.

When used in a LAN networking environment, the personal computer 20 is connected to the local network 51 through a network interface or adapter 53. When used in a WAN

networking environment, the person computer 20 typically includes a modem 54 or other means for establishing communications over the WAN 52. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network

connections shown are exemplary and other means of establishing a communications link between the computers may be used.

In the description that follows, the invention will be described with reference to acts and symbolic representations of operations that are performed by one or more computers, unless indicated otherwise. As such, it will be understood that such acts and operations, which are at times referred to as being computer-executed, include the manipulation by the processing unit of the computer of electrical signals representing data in a structured form. This manipulation transforms the data or maintains it at locations in the memory system of the computer, which reconfigures or otherwise alters the operation of the computer in a manner well understood by those skilled in the art. The data structures where data is maintained are physical locations of the memory that have particular properties defined by the format of the data. However, while the invention is being described in the foregoing context, it is not meant to be limiting as those of skill in the art will appreciate that various of the acts and operation described hereinafter may also be implemented in hardware.

The present invention is directed to a way to enable a software developer to reuse existing code written for creating objects of a base class to create objects of a later-developed class derived from the base class. To allow an appreciation of the significant advantages provided by the invention, the

5 development process, a software application 68 may include one  
or more existing reusable modules. Each of the existing  
modules may include instructions to create objects of various  
classes. For illustration purposes, FIG. 2 also shows a  
structure 69 of objects created in the dynamic memory by a  
10 reusable module 70, in which an object of a class M creates an  
objects of a class H, which creates objects of a class F,  
which in turn create objects of a class A. Although the  
hierarchy of objects in this example is simple, those skilled  
in the art will appreciate that the hierarchy of objects of a  
15 module in a large software application can become very  
complex.

At a later stage of the software development after the reusable modules are generated, new classes inheriting from those classes used in the existing modules may be designed to provide refined functionality and features. Such new classes are intended to be used in the program where the old classes were used. In other words, objects of the new classes are to be created where the existing code would create objects of the old classes. With conventional object-orienting programming languages, however, it would be very difficult to reuse the existing modules to create objects of the new classes. For instance, a new class B derived from the old class A may be

developed for use in place of the class A. The replacement may be selective, i.e., not all objects of the class A are to be replaced by objects of the class B. FIG. 2 shows, for example, a revised in-memory structure 80 of objects that is intended to be created by the use of the new class B in the new code. As shown in this desired new memory structure, an object 72 of the class A in the original hierarchy 69 is replaced with an object 78 of the new class B.

With conventional object-programming techniques, in order to achieve the new hierarchy, the source code 82 of the existing module 70 would have to be modified to include the definition of the class B, and selected instructions in the code to create objects of A would have to be changed to create objects of B. If the programming language is a compiled language, such as C++, the modified source code then has to be recompiled to generate a revised module. The need to perform extensive editing and recompiling makes it difficult to incrementally develop a software application by introducing new classes inheriting from previously developed classes. This problem is especially significant for large software projects using the object-oriented approach, where the flexibility of incremental development is needed the most, due to the complexity of the object hierarchies and the large number of reusable modules involved.

It will be appreciated that this replacement of objects of a base class with objects of a derived class is fundamentally different from the operation of overriding a

00535090-033000

function of a base class with a function of a derived class.

In the existing C++ language, such "polymorphism" with respect to functions is through the use of "virtual functions." For virtual functions to work, both the base class and the derived class have to be defined in the source code of the executable module. In other words, the executable code has to know how to create objects of both the base class and the derived class. In contrast, in the example given in FIG. 2, the replacement class B may be designed after the reusable module 70 was generated.

In accordance with the present invention, the difficulties in reusing existing code to handle new classes are avoided through the use of replaceable classes and virtual constructors. The class replacement in accordance with the invention enables dynamic creation of objects of a new derived class instead of objects of a base class during execution of the existing code in the program. Thus, the use of replaceable classes makes the existing executable modules fully reusable in applications where objects of the newly developed derived classes are to be created. As a result, there is no need to edit and recompile the source listing of the existing executable modules to accommodate the new class.

Also, in many cases, it is very important to supply reusable modules without the corresponding source listings for various reasons, such as to save storage space or to avoid disclosure of proprietary technologies involved in the program sources or the tools used to build the program. Using

5       As described in greater detail below, the class  
replacement may be implemented with both compiled languages  
and interpreted languages. Moreover, with a compiled  
language, the class replacement may be implemented either with  
an existing compiler or with a modified compiler that provides  
0 new features for supporting the class replacement.

Referring to FIG. 3, a class A that is intended to be replaced later by a derived class is declared to be replaceable in a source listing 84. The source listing 84 containing the definition of A and instructions to create objects of A is then compiled into an executable module 86. Later, a class B intended to replace the class A is declared to be a replacement of the class A in a new source listing 88. The source listing 88 containing the definition of the class B and instructions to use the reusable module 86 is compiled into another module 90. The module 90 containing the new code is then combined with other modules, including the reusable module 86, to form an executable program 94. During the execution of the program 94, the new code in the module 90 calls special instructions to indicate, or register, the class replacement relationship between the classes A and B in a

5

10

In a preferred embodiment, new compiler-supported features of the C++ language for handling the layer of



5 Specifically, to indicate that a class is replaceable, its  
constructor is declared to be "virtual." In the existing C++  
language syntax, multiple constructors can be implemented in a  
class. Declaring a given constructor as "virtual" means that  
an attempt to create an object by calling this particular  
10 constructor may actually result in the creation of an instance  
of a yet unknown class that inherits from the class owning the  
virtual constructor. Once a base class is declared to be  
replaceable, a derived class that inherits from the base class  
can overwrite the "virtual" constructor of the base class.  
15 This overwriting constructor will be called each time an  
instance of the new class is to be created instead of an  
instance of the base class.

20

25

By way of example, for a replaceable base class A and a replacement class B inheriting from the class A, the following partial source code shows how the constructors of the two classes are declared:

```

class A
{
    virtual A(int i); // the class A can be replaced
}

class B : public A
{
    virtual A(int i); // replacement constructor as A
    virtual B(int i, char *s); // constructor used
                                // by the new code.
}

```

20 In the above source listing, the statement "virtual A(int  
i);" in the definition of the class A indicates that the class  
A is replaceable. The statement "virtual A(int, i);" in the  
definition of the class B indicates that the class B may be  
used as a replacement for the class A. Here the class B has  
25 another constructor, B(int i, char s), so that it can be  
created from the new code with more information. Since this  
constructor is also declared as virtual, it is also  
replaceable, i.e., an attempt to create an instance of class B  
using this constructor may actually create an instance of  
30 another replacement class derived from B. As shown in FIG. 3,  
the class definitions for the classes A and B may be contained  
in separate source files, which are compiled into different  
object files. The compiler 98 emits the code for the virtual

In accordance with the embodiment, for each class with a virtual constructor, the compiler 98 emits a "Creation

5 Information" block in the initialized data segment of the .obj  
file, along with the virtual function tables ("vtables") for  
the virtual functions of that class. The Creation Information  
block contains the size of the object and the offsets of all  
vtables inside the object. For a replacement class, the  
10 compiler also emits a CLSREF\_DATA record for each of its base  
classes with virtual constructors, which contains a pointer to  
the Creation Information record and the offset of the base  
class in the instance of the current class. Also, for each  
class, a CLSREF\_DATA record to create an object of the class  
15 as itself is always emitted. This CLSREF\_DATA record has a  
vtable offset always set to 0. As will be described below,  
the CLSREF\_DATA and Creation Information records of the  
replaceable and replacement classes are used to provide the  
layer of indirection in object creation for enabling class  
20 replacement.

To illustrate the data emitted by the compiler into the object files to support class replacement and how such data are interrelated and used, FIGS.4-8 provide an example in which a class C inheriting from a conventional (i.e., non-replaceable) class B1 and a replaceable class B2. The new class C can be used as a replacement for the class B2. Turning first to FIG. 4, the class C 100 is described in a new

5 "classes.cxx", and a "functions.cxx" file that contains instructions to create instances of the class B2. More detailed exemplary partial source code for these three files of the old source listing 104 is provided below.

```

10 // classes.hxx
    class B1
    {
    public:
        // non-virtual constructor
        //=>class B1 can't be replaced
        B1(int x);

        virtual void XMove()

20 protected:
        int m_x;
    };

    class B2
25 {
    public:
        // class B2 can be replaced
        virtual B2(int y)

30         virtual void YMove()

        protected:
            int m_y;
    };

35 // classes.cxx
#include classes.hxx

    B1::B1(int x)
40 { m_x=x; };

    void B1::XMove()
    { m_x++; };

45 B2::B2(int y)
    { m_y=y; };

```

```

5 // functions.cxx
  #include classes.hxx

  SomeFunction(...)
  {
10      ...
      B2* pB2=NULL;
      ...
      pB2=new B2();
      ...
15  }

```

```

25 // newclass.hxx
#include classes.hxx

class C: public B1, public B2
{
public:

30 // replacement constructor "as B2"
virtual B2(int y)

// class C can be replaced too
virtual C(int x, int y, int z)

35 virtual void MoveAll()

private:
    int m_z;
40 };
...

// newclass.cxx
#include newclass.hxx

45 C::B2(int y)
{m x=10; m y=y; m z=10};

```

```
void C::MoveAll()
```

```
// main.cxx
```

```
10    main()
```

{

```
CLSREF clsrefOld=classref(B2,B2);
```

```
15  replace class(clsrefOld, clsrefNew);
```

```
// Reuse the old code functions,
```

```
20    SomeFunction();
```

• • •

```
25 } replace_class (NAME, 2, /
```

,

```
30 the base class B1 is not replaceable because its constructor
```

35 for supporting class replacements, i.e., they are the same as

without class replacement. This allows the usage of the

instances of the replaceable classes from functions compiled

with the existing compilers. Since both the classes B1 and B2

```
40  have virtual functions, their objects 120 and 122 have
```

5  
10

15

20

25

5 CLASS\_INFO\_B2 143 and a CLSREF\_DATA\_B2\_B2 record 144, which describes how to create an instance of the class B2. Since the class B2 does not inherit from any classes with virtual constructors, it could not be used as a replacement class, and thus no more CLSREF\_DATA records are emitted.

The Creation Information block 155 for the class C includes a field 156 for the size of an object of C (20 bytes



5

10

15

20

25

```
typedef CLSREF *CLSREF_DATA;
```

30

When the compiler sees a "call" to this function, it will emit instructions to return a CLSREF pointer to the proper

processing the class definition. Note that the compiler will actually emit the symbol here and the linker will later

replace this symbol with the real pointer to the CLSREF\_DATA record in the initialized data segment. As an example, from

FIG.4, calling `classref(C,B2)` will return the pointer to the `CLSREF_DATA_C_B2` record 162 as shown in FIG. 7, while calling `classref(C,C)` will return the pointer to the `CLSREF_DATA_C_C` record 161.

In a preferred embodiment, the compiler support for class replacement also includes two special implementations of the

"new" operator that function differently from the conventional "new" operator of the C++ language. An object of a

replaceable class can be created by using either the operator "static new", which accepts the CLSREF variable type, or the

operator "new", which has the usual syntax of the "new"

operator of the conventional C++ language but may result in the creation of an object of some replacement type. As

described in greater detail below, the implementation of the "new" operator for class replacement utilizes the

implementation of the "static new" operator.

```
<Class*> static_new <CLSREF> BaseClass(...);
```

```
10  "BaseClass" as itself will be used instead.
```

the code, it will emit instructions in the .obj file to:

- 4) Call the virtual constructor of "BaseClass" which matches the argument list. Since the constructor is

5

"BaseClass" instance.

10

```
CLSREF ref=classref(C,B2);
```

15

For the line "CLSREF ref=classref(C,B2);"

- 20

25

- Using the CLSREF\_DATA record pointed to by the local variable "ref", allocate memory space of the size indicated in the associated Creation Information record. In this example, the variable "ref" points to the

5

- 25

The program may now continue, using the above instance of class C by the B2 pointer as usual.

In a preferred embodiment, the implementation of the operator "delete" is also changed for supporting class replacement. Because of the dynamic class replacement, it should not be assumed that the pointer to the object to be deleted is the same as the pointer of the memory block allocated for the object. To that end, additional code is emitted in all virtual destructors to return the pointer to the real object (i.e., the pointer "this"). The delete operation will then first call the virtual destructor (as with existing compilers), and then use the returned pointer to delete the memory block, instead of using the pointer to the object passed as an argument. Note that the returning of "this" pointer from the destructors is hidden from the program - "return" should not be used in the source code of the destructors as in the current C++ standard. As mentioned above, if no destructor is declared for a given class, the compiler will generate one just to return the pointer "this".

It should be noted that the creation of class instances based on a CLSREF variable as described above is not yet automatic "class replacement", because in order to use the static\_new operator, the program has to know explicitly the CLSREF value of the new class. In contrast, the automatic class replacement is carried out by dynamically passing the correct CLSREF value for the class of the object to be created. This operation is enabled by registering class

```
5      typedef struct _CLASS_REPLACEMENT
        {
            CLSREF clsrefClass;
            CLSREF clsrefReplacement;
        } CLASS_REPLACEMENT;
```

15        For the registration of class replacement, a compiler-supported function is provided:

For example, when the compiler processes a source line:

20 it emits code that will add a CLASS\_REPLACEMENT record to the  
list 166 of the class replacements in the memory space for the  
current thread, as illustrated in FIG. 8. As shown in FIG. 8,  
for the present example, this CLASS\_REPLACEMENT record 167  
will be initialized to the address of CLSREF\_DATA\_B2\_B2 record  
25 144 (FIG. 6) describing the creation of class B2 as itself,  
and the clsrefReplacement will be initialized to the address  
of the CLSREF\_DATA\_C\_B2 record 162 (FIG. 7) describing the  
creation of C as B2. In a preferred embodiment, another  
compiler-supported function is provided for un-registering the

class replacement, thereby allowing class replacement to be dynamically switched on and off during program execution.

The dynamic replacement of classes is invoked by using the operator "new," the implementation of which has been  
 5 modified to support class replacements. For example, when the compiler processes the line

```
pB2 = new B2(3);
```

the following code is emitted into the .obj file:

- A call to a compiler-supported function that searches the  
 10 list of replacement records for the current thread. The CLSREF variable pointing to the CLSREF\_DATA record of the class B2 created as itself (i.e., the CLSREF\_DATA\_B2\_B2 record 144 in FIG.6) will be passed to this function as an argument. If the function finds a record in the  
 15 registered replacement list for the base class, the function will return the CLSREF variable pointing to the CLSREF\_DATA record for the replacement class (in this case the CLSREF\_DATA\_C\_B2 record 162 in FIG. 7). On the other hand, if no replacement record for B2 is found, the  
 20 function simply returns the CLSREF argument passed to it.
- A call to static\_new to create an object using the CLSREF returned above.

If no replacement for the class B2 is registered, the program will obtain a new instance of the class B2 as  
 25 requested. If, on the other hand, the replacement of B2 with C is registered, an object of C will be created instead. In this way, class replacement is accomplished.

0055900-0606E560



5 example COM creates the objects based on CLSREF and the COM components may be replaced, but at very high cost - searching the registry for all CLSREF-s known in the system, in order to create a single instance. On the contrary, in both the embodiments described above, only the registered replacements  
10 are searched. Basically, the performance of "new" will be almost the same as without the class replacement support, unless many thousands of class replacements are registered (a very unusual case).

Although the embodiment described above proposes new compiler support for replaceable classes, class replacement according to the invention may be implemented without the need to modify existing compilers. By way of example, one such

20 embodiment is described below.

```
class ClassRef
{
    friend class CObjCreator;
```

5

The CLSREF type in this embodiment is declared as:

```
typedef ClassRef* CLSREF;
```

10

For each replaceable class, an object derived from the ClassRef class is created to implement the CreateInstance() method for creating an object of that replaceable class. This is achieved by using the following template:

15

20

25

30

To identify a class as replaceable, a special macro `REPLACEABLE_CLASS(Class,BaseClass)` is used, which hides the usage of the template above:

35

40

40



5

10 where clsrefOld is a pointer to the Classref object for the old class that is to be replaced, and clsrefNew is a pointer to the ClassRef object for the new class that replaces the old class. This causes the CObjCreator object to add a record in its internal list of replacements. The replacement record is  
15 a structure of type CLASS\_REPLACEMENT defined in the previous embodiment. Note that the replacement is valid only for the objects created by this instance of CObjCreator, until the replacement is changed or all the replacements are reverted.

In this embodiment, in order to keep the implementation of CObjCreator independent of the particular syntax of the constructors of the replaceable classes, it is expected that their actual constructors will have no arguments. The data members of the new instance created will be instantiated later by some virtual method. This method is then Virtual Constructor, in the sense that the object data is uninitialized and the object could not be used before calling this method. It is up to the newly created object to call the initializing methods for its base classes.

The following description provides an example to illustrate the mechanism of class replacement in this embodiment. This example involve a class CBase1 and a class CBase2 that do not inherit from any classes, a class

5 CInherited that inherits from both CBase1 and CBase2 as public, a class CReplace1 that inherits from CBase1 as public, a class CReplace2 that inherits from CInherited as public, and a class Creplace3 that inherits from Cbase2 as public. These classes have different implementations of the method: virtual

10 void Init(); which prints the name of the respective class.

At the global level, the classes are declared in the source listing to be replaceable as follows:

```

15 // At global level
REPLACEABLE_CLASS(CBase1,CBase1);
REPLACEABLE_CLASS(CBase2,CBase2);
REPLACEABLE_CLASS(CInherited,CInherited);
REPLACEABLE_CLASS(CReplace1,CBase1);
REPLACEABLE_CLASS(CReplace2,CInherited);
20 REPLACEABLE_CLASS(CReplace3,CBase2);

```

20 As described above, these declarations cause the creation of the respective ClassRef objects: CLSREF\_CBase1\_CBase1, CLSREF\_CBase2\_CBase2, CLSREF\_CInherited\_CInherited, CLSREF\_CReplace1\_CBase1, CLSREF\_CReplace2\_CInherited, and

25 CLSREF\_CReplace3\_CBase3 to be created. In this example, an object named Creator of the class CObjCreator is used to create objects of these replaceable classes. Instructions to replace CBase1, CBase2, and CInherited with CReplace1, CReplace3, and CReplace2, respectively, are provided by using

30 the Creator.ReplaceClass function in the "main" portion of the code as follows:

000000" 0606E960

```

// in main()
CObjCreator Creator;
CBase1 *pBase1=NULL;
5 CBase2 *pBase2=NULL;
CInherited *pInherited=NULL;

Creator.ReplaceClass(
    CLSREF_CBase1_CBase1,CLSREF_CReplace1_CBase1);
10 Creator.ReplaceClass(
    CLSREF_CInherited_CInherited,CLSREF_CReplace2_CInherited);
Creator.ReplaceClass(
    CLSREF_CBase2_CBase2,CLSREF_CReplace3_CBase2);
15

// In some reusable function
Creator.Create(CLSREF_CBase1, (LPVOID*)&pBase1);
pBase1->Init();

20 Creator.Create(CLSREF_CInherited, (LPVOID*)&pInherited);
pInherited->Init();

Creator.Create(CLSREF_CBase2, (LPVOID*)&pBase2);
pBase2->Init();
25

```

Due to the class replacement, the objects created by the Creator.Create function are those of the replacement classes CReplace1, CReplace3, and CReplace2, instead of those of the replaced classes CBase1, CBase2, and CInherited, even though they appear as arguments for the Creator.Create function. If the Init functions are implemented to print the name of the class, the output of the program would be:

```

CReplace1
35 CReplace2
CReplace3

```

Although in this example the Create function is invoked in the "main" portion of the program, it will be appreciated that the instructions to create objects of the replaceable classes may be included in an existing reusable module that

5

## Class Replacement with An Interpreted Language

10

15

20

25

5 Therefore, the invention as described herein contemplates all such embodiments as may come within the scope of the following claims and equivalents thereof.